MANUAL DE ARQUITECTURA

Sistema de Tiendas Online

Pamplona, 17 de Junio 2025

Versión 1.0

Autores

Fernanda García

Bayron Cañas

Descripción General

Tiendatek es una plataforma de e-commerce diseñada para ofrecer una experiencia de compra simple y directa a los clientes, y un conjunto de herramientas de gestión robustas para los empleados. La aplicación permite a los usuarios registrarse con roles diferenciados (cliente o empleado), explorar un catálogo de productos con imágenes, gestionar un carrito de compras persistente y realizar pedidos.

Para el personal interno, provee un panel de control seguro para la gestión completa del ciclo de vida de los productos, incluyendo la subida de imágenes, y la capacidad de auditar tanto las ventas como las actividades internas del sistema.

El objetivo de esta arquitectura es sentar una base tecnológica sólida, segura y escalable para futuras iteraciones del proyecto, priorizando el rendimiento.

Tecnologías Utilizadas (Backend)

- Lenguaje: JavaScript
- Entorno de ejecución: Node.js (v18.x o superior)
- Framework web: Express.js
- Base de datos: PostgreSQL
- Plataforma de Despliegue: Render (como "Web Service")
- Almacenamiento de Archivos: Cloudinary para la gestión y persistencia de imágenes.
- Cliente SQL: pg (node-postgres)
- Autenticación: JSON Web Tokens (JWT) y bcryptjs.
- Seguridad de Contraseñas: bcryptjs para el hasheo de contraseñas.
- Gestión de Subida de Archivos: multer para procesar la subida de imágenes.
- Dependencias principales:
  - ✓ express, cors, helmet: Para la estructura del servidor y la seguridad.
  - ✓ jsonwebtoken, bcryptjs: Para la autenticación y seguridad de usuarios.
  - ✓ pg: Para la conexión con la base de datos PostgreSQL.
  - ✓ dotenv: Para la gestión segura de variables de entorno.
  - ✓ multer: Para el manejo de archivos subidos.
  - ✓ nodemon (desarrollo): Para reiniciar el servidor automáticamente.

Tecnologías Utilizadas (Frontend)

- Tecnologías Principales: HTML5, CSS3, JavaScript (Vanilla JS, ES6+).
- Cliente HTTP: Fetch API nativa del navegador.
- Gestión de Estado Local: Web Storage API (localStorage).
- Formato de La Moneda: Intl.NumberFormat API para localización a Pesos Colombianos (COP).
- Plataforma de Despliegue: Netlify.

Arquitectura Implementada: MVC Desacoplado

Descripción del Patrón

Elegimos el patrón arquitectónico **MVC (Modelo–Vista–Controlador)** porque nos permitió organizar el sistema de forma clara y modular. Separar la lógica del negocio, la interfaz de usuario y el control de flujo facilitó el desarrollo colaborativo y el mantenimiento del código.

Además, MVC es ampliamente utilizado en aplicaciones web modernas, y se adapta perfectamente a tecnologías como React para la vista y JavaScript en el backend. Gracias a esta estructura, pudimos trabajar de forma ordenada, reutilizar componentes y facilitar futuras mejoras.

Ventajas

- Organización y Mantenibilidad: Nos ha permitido separar la lógica de las rutas (Controladores) de la lógica de acceso a datos (Modelo), haciendo el código mucho más legible y fácil de mantener a medida que crece.
- Desarrollo en Paralelo: Facilitó que el desarrollo del frontend (index.html, empleado.html) y el backend (Node.js) pudiera ocurrir de forma independiente.
- Reusabilidad del Backend: La misma API que sirve a nuestra web podría, en el futuro, servir a una aplicación móvil nativa sin necesidad de cambios en la lógica de negocio.

Estructura del Proyecto (Solo Backend)

La estructura de directorios del backend está organizada para reflejar la separación de responsabilidades del patrón MVC.

tiendatek-backend/

├── config/

│ └── database.js # Configura y exporta la conexión a PostgreSQL

├── middleware/

│ └── auth.js # Middlewares authenticateToken y requireEmployee

├── routes/

│ ├── auth.js # Endpoints para registro y login

│ ├── logs.js # Endpoint para el historial de actividades

│ ├── orders.js # Endpoints para pedidos e historiales

│ └── products.js # Endpoints CRUD para productos (con subida de imagen)

├── uploads/ # Directorio donde se almacenan las imágenes

├── .env # Variables de entorno (NO versionar)

├── server.js # Punto de entrada de la aplicación Express

└── package.json # Dependencias y scripts

Generated code

Estructura del Proyecto (Solo Frontend)

La estructura del frontend es intencionadamente simple y plana, con toda la lógica contenida dentro de los archivos principales para facilitar el despliegue en cualquier servidor de archivos estáticos.

Raíz del proyecto:

Generated code

```
└── frontend/
    ├── index.html
    └── empleado.html
```

index.html: La vista y lógica para el cliente.

empleado.html: El panel de administración para el empleado.

Componentes Lógicos (Funciones JavaScript): El código dentro de cada archivo está organizado en funciones con responsabilidades claras:

- Funciones de Renderizado: renderProducts(), openCart(), etc., que generan HTML dinámico.
- Funciones de Estado: checkAuth(), saveCartToLocalStorage(), etc., que gestionan la sesión y el estado local.
- Servicio de API: La función apiFetch() centraliza todas las llamadas al backend, estandarizando la comunicación.

Clases, Objetos y Artefactos que Sustentan la Arquitectura

Modelo (Lógica de Datos en routes/ + config/database.js)

- Responsabilidad: Es la única capa que interactúa directamente con la base de datos PostgreSQL. Se encarga de leer, escribir, actualizar y borrar datos.
- Implementación: A través del cliente pg y consultas SQL puras. config/database.js gestiona el pool de conexiones para optimizar el rendimiento.
- Ejemplo: En routes/products.js, la consulta db.query('INSERT INTO productos...') es la ejecución de la lógica del Modelo.

Vista (Archivos index.html y empleado.html)

- Responsabilidad: Presentar los datos al usuario y capturar sus interacciones. No contiene lógica de negocio.
- Implementación: HTML para la estructura, CSS para el estilo y Vanilla JavaScript para la interactividad y la comunicación con la API.

Controlador (Archivos en routes/)

- Responsabilidad: Actuar como intermediario. Recibe las peticiones HTTP desde la Vista, las procesa y llama al Modelo para obtener o manipular datos.
- Implementación: Cada función async (req, res) dentro de nuestros archivos de rutas actúa como un método de un controlador.
- Ejemplo: La función que maneja POST /api/products en products.js controla el flujo para crear un nuevo producto.

Middlewares (middleware/auth.js)

- Responsabilidad: Interceptar peticiones para realizar tareas transversales como la autenticación y la autorización.
- Implementación: authenticateToken verifica el JWT, mientras que requireEmployee comprueba el rol del usuario, protegiendo las rutas de gestión.

Componentes, Funciones y Artefactos que Sustentan la Arquitectura (Frontend)

Páginas (HTML principales)

- Responsabilidad: Representar vistas completas para cada tipo de usuario.
- Función: Integran HTML, CSS y JS para mostrar información y gestionar interacciones.
- Ejemplos:
  - ✓ index.html: Vista para clientes (productos, carrito, pedidos).
  - ✓ empleado.html: Panel de empleados (CRUD productos, ventas, historial).

Componentes de Renderizado (Funciones UI)

- Responsabilidad: Crear y actualizar partes del DOM.
- Función: Transforman datos en HTML dinámico.
- Ejemplos: renderProducts(), openCart(), renderOrderHistory().

Componentes de Estado y Sesión

- Responsabilidad: Mantener el estado global de la app.
- Función: Usan variables JS y localStorage para gestionar autenticación y carrito.
- Ejemplos: login(), logout(), saveCartToLocalStorage().

Servicio de API (apiFetch)

- Responsabilidad: Manejar toda la comunicación con el backend.
- Función: Realiza peticiones HTTP, añade tokens y gestiona respuestas.
- Ejemplo: apiFetch() es reutilizada por todas las funciones que llaman a la API.

Flujo de Comunicación MVC (Ejemplo: Empleado Crea un Producto)

1. Vista: Un empleado llena el formulario en empleado.html y sube una imagen. Al hacer clic en "Guardar", el JavaScript crea un objeto FormData y envía una petición POST a /api/products.

2. Ruta/Controlador: server.js dirige la petición a routes/products.js. El middleware multer procesa y guarda el archivo de imagen.

3. Autorización: Los middlewares authenticateToken y requireEmployee verifican que el usuario sea un empleado válido y con sesión activa.

4. Controlador: La función principal de la ruta se ejecuta. Valida los datos de texto del req.body y obtiene la ruta de la imagen guardada.

5. Modelo: El controlador ejecuta dos consultas INSERT en la base de datos: una en la tabla productos para guardar el nuevo artículo y otra en historialactividades para registrar la acción.

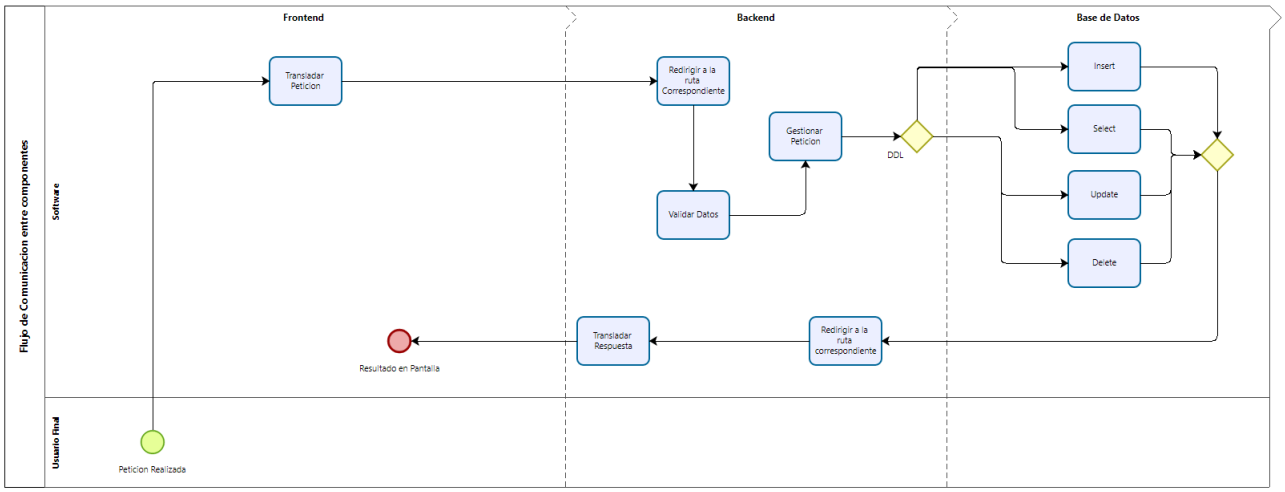
6. Respuesta: La base de datos confirma la inserción. El controlador envía una respuesta JSON de éxito (código 201) de vuelta a la Vista.

7. Vista: El JavaScript del frontend recibe la respuesta, muestra una notificación de éxito y refresca la lista de productos para mostrar el nuevo artículo.

Seguridad y Autenticación

- JWT: Se utiliza para la autenticación sin estado, firmando cada token con una clave secreta (JWT_SECRET) guardada en variables de entorno.
- Bcrypt: Todas las contraseñas se cifran con un "salt" y un algoritmo de hashing robusto antes de ser almacenadas.
- Autorización por Roles: El middleware requireEmployee protege las rutas críticas, asegurando que solo el personal autorizado pueda realizar cambios en el inventario.
- Variables de Entorno (.env): Las claves, contraseñas y configuraciones sensibles se mantienen fuera del código fuente y del control de versiones (Git), siguiendo las mejores prácticas de seguridad.

Diagrama la solución técnica



Conclusión

El proyecto "Tienda-Tek" logró consolidarse como una solución web moderna, robusta y escalable. Superó desafíos clave como el almacenamiento efímero mediante la integración de Cloudinary, fortaleciendo su arquitectura. La implementación de una estructura desacoplada con servicios especializados (Netlify, Render, Cloudinary) facilitó el mantenimiento y la escalabilidad. Además, se cubrió todo el ciclo de desarrollo, desde la programación hasta el despliegue y resolución de problemas en producción. En conjunto, "Tienda-Tek" queda bien posicionada para futuras mejoras y expansión.Arquitectura Desacoplada Efectiva